UNITED STATES PATENT APPLICATION

FOR

HARDWARE RECOVERY IN A
MULTI-THREADED ARCHITECTURE

INVENTORS:

STEVEN K. REINHARDT
SHUBHENDU S. MUKHERJEE
JOEL S. EMER

PREPARED BY:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN, LLP
12400 WILSHIRE BOULEVARD
SEVENTH FLOOR
LOS ANGELES, CA 90025-1026

(503) 684-6200

EXPRESS MAIL NO. EV 325528410 US

HARDWARE RECOVERY IN A
MULTI-THREADED ARCHITECTURE

RELATED APPLICATIONS

This U.S. Patent application is related to the following U.S. Patent applications:

(1)   INCREMENTAL CHECKPOINTING IN A MULTI-THREADED

ARCHITECTURE, application number (Attorney Docket No. P15452) filed August 29,

2003; and

(2)   PERIODIC CHECKPOINTING IN A REDUNDANTLY MULTI-

THREADED ARCHITECTURE, application number (Attorney Docket No. P15453),

filed August 29, 2003.

TECHNICAL FIELD

[0001]   The invention relates to multi-threaded computing architectures. More

specifically, the invention relates to devices and techniques for hardware error recovery

in a multi-threaded computing architecture.

BACKGROUND

[0002]   Processors are becoming increasingly vulnerable to transient faults caused by

alpha particle and cosmic ray strikes. These faults may lead to operational errors referred

to as "soft" errors because these errors do not result in permanent malfunction of the

processor. Strikes by cosmic ray particles, such as neutrons, are particularly critical

because of the absence of practical protection for the processor. Transient faults

currently account for over 90% of faults in processor-based devices.

[0003] As transistors shrink in size the individual transistors become less vulnerable to cosmic ray strikes. However, decreasing voltage levels that accompany decreasing transistor sizes and the corresponding increase in transistor count results in an exponential increase in overall processor susceptibility to cosmic ray strikes or other causes of soft errors. To compound the problem, achieving a selected failure rate for a multi-processor system requires an even lower failure rate for the individual processors. As a result of these trends, fault detection and recovery techniques, typically reserved for mission-critical applications, are becoming increasing applicable to other processor applications.

[0004] Several terms are commonly used when discussing processor errors and error recovery. A Failure in Time (FIT) refers to an error rate of one failure in one billion (10^9) hours. Mean Time Between Failure (MTBF) is the time between failures caused by soft errors. MTBF requirements are typically expressed in years. For example, one FIT equals a MTBF of 114,155 years: $114,155 = \dfrac{10^9}{(24*365)}$.

[0005] Silent Data Corruption (SDC) occurs when errors are not detected and may result in corrupted data values that can persist until the processor is reset. The SDC Rate is the rate at which SDC events occur. Detected, unrecoverable errors (DUE) are errors that are detected, for example, by using parity checking, but cannot be corrected. The rate of these errors is referred to as the DUE rate.

[0006] For example, publicly available documents from IBM (D.C. Bossen, "CMOS Soft Errors and Server Design," IBM Server Group, Reliability Physics Tutorial Notes, Reliability Fundamentals, April 2002.), specify 25 years MTBF for DUE and 1000 years MTBF for SDC. These specifications are for single-processor systems. Application to a

multi-processor system results in more stringent specifications for individual processors. It is becoming increasingly difficult to meet SDC and DUE FIT specifications because the neutron FIT contribution of latches is increasing. Other components, for example, most SRAM cells, either can be protected via interleaved parity or error correcting codes or do not provide significant contribution to the overall FIT rate.

[0007] The FIT rate of latches consists of two parts: the raw FIT rate and a derating factor. The raw FIT rate can be computed using circuit models and currently ranges between 0.001 and 0.01 per latch. The derating factor is the fraction of faults that lead to errors. Typically, the derating factor is 10%. See, for example, Eugene Normand, "Single Event Upset at Ground Level," IEEE Transactions on Nuclear Science, Vol. 43, No. 6, December 1996 and Y. Tosaka, et al., "Impact of Cosmic Ray Neutron Induced Soft Errors on Advanced Submicron CMOS Circuits," VLSI Symposium on VLSI Technology Digest of Technical Papers, 1996. Using the specifications set forth above as a further example, in a 64-processor system, each processor can have only approximately 1,800 latches. However, designing a complex, high-performance processor core with only 1,800 latches is extremely difficult.

[0008] Fault detection support can reduce a processor's SDC rate by halting computation before faults can propagate to permanent storage. Parity, for example, is a well-known fault detection mechanism that avoids silent data corruption for single-bit errors in memory structures. Unfortunately, adding parity to latches or logic in high-performance processors can adversely affect the cycle time and overall performance. Consequently, processor designers have resorted to redundant execution mechanisms to detect faults in processors.

[0009]   Current redundant-execution systems commonly employ a technique known as "lockstepping" that detects processor faults by running identical copies of the same program on two identical lockstepped (cycle-synchronized) processors. In each cycle, both processors are fed identical inputs and a checker circuit compares the outputs. On an output mismatch, the checker flags an error and can initiate a recovery sequence. Lockstepping can reduce a processor's SDC FIT by detecting each fault that manifests at the checker. Unfortunately, lockstepping wastes processor resources that could otherwise be used to improve performance.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings in which like reference numerals refer to similar elements.

Figure 1 is a block diagram of one embodiment of a redundantly multithreaded architecture.

Figure 2 is a block diagram of one embodiment of a simultaneous and redundantly threaded architecture.

Figure 3 is a block diagram of one embodiment of a chip-level redundantly threaded architecture.

Figure 4 illustrates minimum and maximum slack relationships for one embodiment of a simultaneous and redundantly threaded architecture.

Figure 5 is a conceptual flow of a piecewise deterministic system.

Figure 6 is a flow diagram of one embodiment of recovery during normal operation.

Figure 7 is a flow diagram of one embodiment of fault recovery during checkpoint creation.

Figure 8 is a conceptual illustration of a load value queue used for recovery.

Figure 9 is a block diagram of an electronic system that can provide an environment for multithreaded processors.

DETAILED DESCRIPTION

[0010]   Methods and apparatuses for hardware recovery in a multithreaded architecture are described. In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the invention. It will be apparent, however, to one skilled in the art that the invention can be practiced without these specific details. In other instances, structures and devices are shown in block diagram form in order to avoid obscuring the invention.

[0011]   In one embodiment, a redundant multithreaded (RMT) system provides fault detection in a more efficient manner than traditional techniques such as lockstepping. The RMT processor detects hardware faults by running corresponding copies of the same thread with identical inputs and comparing the outputs for mismatches. By making the domain of checking the processor, the outputs of the processors, for example, memory writes (address and data) are checked.

[0012]   Transparent hardware recovery requires a fault-free checkpointed state of the system so that on an error the processor can reload the fault-free state and repeat execution. In one embodiment, the checkpointed state of a system includes both architectural register values and memory. The checkpoint consists of a copy of the architectural register file values at the time the checkpoint is generated. In one embodiment, checkpointing also includes an ordered log of responses from memory so that the responses can be repeated to simulate a complete checkpoint of the memory.

[0013]   Described in greater detail below are techniques for periodic checkpointing of the architectural register file and logs of external (to the processors) inputs, such as load values, to allow transparent recovery from an error. When a processor detects an error on

a store instruction (e.g., via an output comparison mechanism), the processor reloads the state from the last checkpoint, repeats the external inputs from the log, but suppresses outputs to memory because those outputs have been previously committed. In one embodiment, this technique uses a log of external inputs to decouple the committing of output values from register checkpointing, allowing higher performance and uses input replication mechanisms to serve as the log of external inputs.

Sphere of Replication

[0014]   **Figure 1** is a block diagram of one embodiment of a redundantly multithreaded architecture. In a redundantly multithreaded architecture faults can be detected by executing two copies of a program as separate threads. Each thread is provided with identical inputs and the outputs are compared to determined whether an error has occurred. Redundant multithreading can be described with respect to a concept referred to herein as the "sphere of replication." The sphere of replication is the boundary of logically or physically redundant operation.

[0015]   Components within sphere of replication 130 (e.g., a processor executing leading thread 110 and a processor executing trailing thread 120) are subject to redundant execution. In contrast, components outside sphere of replication 130 (e.g., memory 150, RAID 160) are not subject to redundant execution. Fault protection is provide by other techniques, for example, error correcting code for memory 150 and parity for RAID 160. Other devices can be outside of sphere of replication 130 and/or other techniques can be used to provide fault protection for devices outside of sphere of replication 130.

[0016]   Data entering sphere of replication 130 enter through input replication agent 170 that replicates the data and sends a copy of the data to leading thread 110 and to

trailing thread 120. Similarly, data exiting sphere of replication 130 exit through output comparison agent 180 that compares the data and determines whether an error has occurred. Varying the boundary of sphere of replication 130 results in a performance versus amount of hardware tradeoff. For example, replicating memory 150 would allow faster access to memory by avoiding output comparison of store instructions, but would increase system cost by doubling the amount of memory in the system.

[0017] In general, there are two spheres of replication, which can be referred to as "SoR-register" and "SoR-cache." In the SoR-register architecture, the register file and caches are outside the sphere of replication. Outputs from the SoR-register sphere of replication include register writes and store address and data, which are compared for faults. In the SoR-cache architecture, the instruction and data caches are outside the sphere of replication, so all store addresses and data, but not register writes, are compared for faults.

[0018] The SoR-cache architecture has the advantage that only stores (and possibly a limited number of other selected instructions) are compared for faults, which reduces checker bandwidth and improves performance by not delaying the store operations. In contrast, the SoR-register architecture requires comparing most instructions for faults, which requires greater checker bandwidth and can delay store operations until the checker determines that all instructions prior to the store operation are fault-free. The SoR-cache can provide the same level of transient fault coverage as SoR-register because faults that do not manifest as errors at the boundary of the sphere of replication do not corrupt the system state, and therefore, are effectively masked.

[0019] In order to provide fault recovery, each instruction result should be compared to provide a checkpoint corresponding to every instruction. Accordingly, the SoR-register architecture is described in greater detail herein.

Overview of Simultaneous and Redundantly Threaded Architecture

[0020] **Figure 2** is a block diagram of one embodiment of a simultaneous and redundantly threaded architecture. The architecture of Figure 2 is a SoR-register architecture in which the output, or result, from each instruction is compared to detect errors.

[0021] Leading thread 210 and trailing thread 220 represent corresponding threads that are executed with a time differential so that leading thread 210 executes instructions before trailing thread 220 executes the same instruction. In one embodiment, leading thread 210 and trailing thread 220 are identical. Alternatively, leading thread 210 and/or trailing thread 220 can include control or other information that is not included in the counterpart thread. Leading thread 210 and trailing thread 220 can be executed by the same processor or leading thread 210 and trailing thread 220 can be executed by different processors.

[0022] Instruction addresses are passed from leading thread 210 to trailing thread 220 via instruction replication queue 230. Passing the instructions through instruction replication queue 230 allows control over the time differential or "slack" between execution of an instruction in leading thread 210 and execution of the same instruction in trailing thread 220.

[0023] Input data are passed from leading thread 210 to trailing thread 220 through source register value queue 240. In one embodiment, source register value queue 240

replicates input data for both leading thread 210 and trailing thread 220. Output data are passed from trailing thread 220 to leading thread 210 through destination register value queue 250. In one embodiment, destination register value queue 250 compares output data from both leading thread 210 and trailing thread 220.

[0024]   In one embodiment, leading thread 210 runs hundreds of instructions ahead of trailing thread 220. Any number of instructions of "slack" can be used. In one embodiment, the slack is caused by slowing and/or delaying the instruction fetch of trailing thread 220. In an alternate embodiment, the slack can be caused by instruction replication queue 230 or an instruction replication mechanism, if instruction replication is not performed by instruction replication queue 230.

[0025]   Further details for techniques for causing slack in a simultaneous and redundantly threaded architecture can be found in "Detailed Design and Evaluation of Redundant Multithreading Alternatives," by Shubhendu S. Mukherjee, Michael Kontz and Steven K. Reinhardt in *Proc. 29th Int'l Symp. on Computer Architecture*, May 2002 and in "Transient Fault Detection via Simultaneous Multithreading," by Steven K. Reinhardt and Shubhendu S. Mukherjee, in *Proc. 27th Int'l Symp. on Computer Architecture*, June 2000.

Overview of Chip-Level Redundantly Threaded Architecture

[0026]   **Figure 3** is a block diagram of one embodiment of a chip-level redundantly threaded (CRT) architecture. As with the SRT architecture described above, the CRT architecture executes loosely synchronized redundant threads, which enable lower checker overhead and reduces or eliminates cache miss and misspeculation penalties in the trailing thread. As in lockstepping, the two redundant thread copies execute on

separate processor cores. That is, the two redundant threads are not multiplexed as different thread contexts on a single processor core as in the SRT architecture. This allows the CRT architecture to provide the same transient and permanent fault protection as lockstepping.

[0027]   Leading thread 310 executed by processor 300 and trailing thread 315 executed by processor 350 represent a first set of corresponding threads (Thread A) that are executed with a time differential so that leading thread 310 executes instructions before trailing thread 315 executes the corresponding instruction. Similarly, leading thread 325 executed by processor 350 and trailing thread 320 executed by processor 300 represent a second set of corresponding threads (Thread B) that are executed with a time differential so that leading thread 325 executes instructions before trailing thread 320 executes the corresponding instruction.

[0028]   Instructions to be executed by trailing thread 315 are passed from leading thread 310 to trailing thread 315 through instruction replication queue 330. Input data are passed from leading thread 310 executed by processor 300 to trailing thread 315 executed by processor 350 through source register value queue 332. In one embodiment, source register value queue 332 replicates input data for both leading thread 310 and trailing thread 315. Output data are passed from trailing thread 315 to leading thread 310 through destination register value queue 334. In one embodiment, destination register value queue 334 compares output data from both leading thread 310 and trailing thread 315.

[0029]   Instructions to be executed by trailing thread 320 are passed from leading thread 325 to trailing thread 320 through instruction replication queue 380. Input data are passed from leading thread 325 executed by processor 350 to trailing thread 320 executed

by processor 300 through source register value queue 382. In one embodiment, source register value queue 382 replicates input data for both leading thread 325 and trailing thread 320. Output data are passed from trailing thread 320 to leading thread 320 through destination register value queue 384. In one embodiment, destination register value queue 384 compares output data from both leading thread 320 and trailing thread 325.

[0030] Instruction queues 330 and 380, source register value queues 332 and 382, and destination register queues 334 and 384 are illustrated as outside of processors 300 and 350 to indicate the independent nature of physical queue placement. The queues illustrated in Figure 3 can be implemented in processor 300, in processor 350 or in both processor 300 and processor 350.

[0031] When executing single-thread workloads, the CRT architecture performs similarly to lockstepping because the behavior of the leading thread is similar to that of the individual thread in a lockstepped processor. However, with multithreaded workloads, the CRT architecture "cross-couples" the processor cores to increase efficiency and provide improved performance as compared to lockstepping. For example, with two application threads, each core runs the leading thread for one application and the trailing thread for the other application. The resources freed by the CRT architecture on each core from optimizing execution of the trailing thread are then available for the more resource-intensive leading thread execution.

[0032] **Figure 4** illustrates minimum and maximum slack relationships for one embodiment of a simultaneous and redundantly threaded architecture. The embodiment of Figure 4 is a SoR-register architecture as described above. The minimum slack is the total latency of a cache miss, latency from execute to retire, and latency incurred to

forward the load address and value to the trailing thread. If the leading thread suffers a cache miss and the corresponding load from the trailing thread arrives at the execution point before the minimum slack, the trailing thread is stalled.

[0033] Similarly, the maximum slack is latency from retire to fault detection in the leading thread. In general, there is a certain amount of buffering to allow retired instructions from the leading thread to remain in the processor after retirement. This defines the maximum slack between the leading and trailing threads. If the buffer fills, the leading thread is stalled to allow the trailing thread to consume additional instructions from the buffer. Thus, if the slack between the two threads is greater than the maximum slack, the overall performance is degraded.

Overview of Checkpointing and Backward Recovery

[0034] One fault-recovery technique, referred to as "backward recovery", involves restoring a system (e.g., a processor, a computer system) to an earlier fault-free state and re-executing a thread from the restoration point. A copy of the earlier state that can be restored is referred to as a "checkpoint." Backward recovery includes two issues to be addressed to provide transparent recovery: non-deterministic events and regenerated outputs.

[0035] If a non-deterministic event (e.g., an asynchronous interrupt) occurs after the last checkpoint re-execution after a fault may not follow the same path as the original execution. Specifically, if an externally visible output (e.g., a store) was generated along the original execution path prior to the fault, but the re-execution follows a different path that generates a different output, the resulting sequence of outputs, as observed from a reference external to the system, will not be consistent with fault-free execution and

recovery will not be transparent. To prevent this occurrence, a backward recovery system must guarantee at each output operation that any subsequent fault-induced rollback will follow the same execution path up to the point of the output. This is referred to as the "output commit" problem.

[0036]   Even if the re-execution deterministically follows the same path as the original execution, any externally visible output operations after the last checkpoint will be performed again during re-execution. If these output operations are not idempotent, then the re-execution will lead to behavior inconsistent with a fault-free execution and recovery will not be transparent.

[0037]   One solution to both issues (non-deterministic events and regenerated outputs) is to create a new checkpoint automatically with each output operation. The checkpoint then incorporates any non-deterministic events that may have led to the execution of that output operation. If the output operation completes successfully, then any subsequent fault will roll back to the execution point after that operation and the output operation will not be re-executed.

Overview of Log-Based Recovery

[0038]   In general, log-based recovery decouples the fault detection point from the checkpoint validation. This decoupling allows a performance advantage of the RMT architecture while also providing transparent hardware recovery that improves the SER FIT of a processor.

[0039]   Log-based recovery applies to a system that is piecewise deterministic (i.e., a system in which execution consists of deterministic intervals separated by non-deterministic events). An event is considered non-deterministic if execution of the event

is not determined entirely by the state inside the system. The state of a piecewise deterministic system at any point can be recreated given an initial condition and a complete description of all non-deterministic events up to the desired point.

[0040]   **Figure 5** is a conceptual flow of a piecewise deterministic system. Checkpoint 500 corresponds to a point in execution of a thread at which a copy of the architectural register file values is stored. In the description of Figure 5, the instructions of a thread that are executed between checkpoints are referred to as "fragments." For example, executing thread fragment 510 represents the instructions executed between the time of checkpoint 500 and checkpoint 550 and executing thread fragment 560 represents the instructions executed between the time of checkpoint 550 and checkpoint 590.

[0041]   During the time between checkpoint 500 and checkpoint 550 one or more non-deterministic events 520 occur. Non-deterministic events 520 are recorded until the time of checkpoint 550 for potential recovery purposes. During the time between checkpoint 550 and checkpoint 590 non-deterministic events 570 occur and are recorded for potential recovery purposes. At the time of checkpoint 550, the recorded non-deterministic events 520 are no longer required because checkpoint 550 corresponds to a storage of the complete architectural state at the corresponding time.

[0042]   In one embodiment, the non-deterministic events are stored in a log that sequentially records the non-deterministic events. When fault recovery is required, the most recent checkpoint is restored and the non-deterministic events are re-executed to determine the correct state. As described above, a non-deterministic event is an event where execution is not determined entirely within the system (e.g., the sphere of replication to which the executing processor belongs). Non-deterministic events include,

for example, load operations (e.g., memory contents modified external to the sphere of the processor), interrupts (e.g., asynchronous interrupts), or timing-dependent operations (e.g., read operations to on-chip cycle counters).

[0043] The non-deterministic events are maintained in order to provide determinism between the leading and trailing threads. When execution is deterministic, both threads follow the same execution path and capture of the non-deterministic events allows the threads to maintain consistency. Thus, the log, or record, of non-deterministic events between checkpoints can be used for error recovery.

[0044] In order to support error recovery, the checkpoint and the log of non-deterministic events must be fault free and protected against transient faults (e.g., by ECC). After completion and fault-free validation of a checkpoint, the previous checkpoint and any logged non-deterministic events prior to the current checkpoint can be discarded. Checkpoint frequency can be based on, for example, the available log storage and/or recovery latency.

[0045] As discussed above, the RMT architecture provides fault detection based on the SoR-cache architecture. When checkpoints are created the data to be stored are checked for faults. In one embodiment, all new register values from the two redundant threads are compared for a mismatch. In the absence of a mismatch, the register values can be sent to fault-protected (e.g., ECC-protected) checkpoint storage.

[0046] The log of non-deterministic events, for example, loads and asynchronous interrupts, allows the architectural state of the processor at the point a fault is detected to be reproduced starting from the last checkpoint, regardless of the time that has passed since the checkpoint. This decouples the fault detection in SoR-cache (used to verify

data sent from the processor) from fault detection in SoR-register (used to validate checkpoints) because outputs (e.g., stores) can be committed externally without creating a full, validated checkpoint at the time of the output. Consequently, checkpoint validation is not in the critical path of program execution and, therefore, can be performed as a background operation. Also, checkpoints are not required for every output, which decreases the frequency of checkpoint storage.

[0047] **Figure 6** is a flow diagram of one embodiment of recovery during normal operation. During regular execution, outputs from the leading and trailing threads are compared to determine whether an output mismatch has occurred, 610. An output mismatch for data exiting the SoR-cache (e.g., in a store address or value) indicates the presence of a fault.

[0048] In response to the fault, both the leading and trailing thread contexts are flushed, 620, because detection of the thread does not indicate which thread caused the fault. Contexts for both threads are reloaded from the most recent checkpoint, 630. The contexts, or architectural state, corresponding to a check point can be restored from a memory that stores the most recent checkpoint.

[0049] Thread processing for both the leading and trailing thread resumes in recovery mode in which both threads are driven by the logged non-deterministic events, 640. Inputs to the trailing thread are the same for normal operation and recovery. The leading thread is temporarily supplied by the logged non-deterministic events until the execution point at which the fault occurred. In one embodiment, the fault point is determined by maintaining a counter that tracks the number of outputs generated since the previous checkpoint. During recovery, the counter is decremented for each output generated until

the count reaches zero. Other techniques for determining the fault point can also be used. During recovery regenerated outputs are discarded, 650, because those outputs have already been exposed outside of the SoR-cache.

[0050]   **Figure 7** is a flow diagram of one embodiment of fault recovery during checkpoint creation. Checkpoint creation may occur either continuously as a background operation or at periodic intervals. Register values from both threads are compared for mismatches, 710. In the absence of a mismatch, 720, the outputs are committed to external checkpoint storage, 730.

[0051]   If a mismatch occurs, 720, both thread contexts are flushed, 740. Contexts for both threads is reloaded from the most recent checkpoint, 750. Thread processing for both the leading and trailing thread resumes in recovery mode in which both threads are driven by the logged non-deterministic events, 760. During recovery regenerated outputs are discarded, 770. The external checkpoint storage must be capable of storing a full valid checkpoint until a subsequent checkpoint can be fully validated.

[0052]   Faults can also occur during the recovery procedure. In one embodiment, to detect these faults, the outputs of both threads that leave the SoR-cache are compared. In one embodiment, a detected fault during recovery can imply a permanent fault, which is flagged for action by the operating system. Alternatively, both threads can be restarted (e.g., a second time) from the checkpoint with input provided from the event log as described above.

[0053]   **Figure 8** is a conceptual illustration of a load value queue used for recovery. In one embodiment, load value queue 800 is a program-order first in/first out (FIFO)

queue of load values as provided to the leading thread. Loads for the trailing thread obtain values from load value queue 800 instead of from the data cache.

[0054]   In order to provide for recovery, load value queue 800 maintains entries after the trailing thread has consumed the entry and until a subsequent checkpoint is completed. In one embodiment in which checkpoints are frequent, entries are stored in the same physical load value queue as used for input replication. In an alternate embodiment, a larger load value queue can be used where entries are copied from a primary load value queue to a recovery load value queue. Because the performance of recovery may not be critical, the recovery load value queue can be larger, slower and/or farther from the processor core than the replication load value queue.

[0055]   Logically, the load value queue can be separated into replication entries 810 that have not been consumed by the trailing thread and recovery entries 820 that have been consumed by the trailing thread, but have not been incorporated into the latest checkpoint. Queue pointer 830 indicates the boundary between replication entries 810 and recovery entries 820.

[0056]   Asynchronous interrupts can be replicated by logging the point in the instruction stream at which the interrupt is take in the leading thread and generating the interrupt at the same point in the trailing thread, or by forcing the thread copies to synchronize to the same execution point and delivering the interrupt simultaneously. In the former scenario, asynchronous interrupts can be logged and replayed in the same manner as loads. The mechanism for generating interrupts from the log can leverage the logic used to generate interrupts in the trailing thread in normal operation. In the latter scenario, interrupts are not stored and transferred between the two threads, so there is no

mechanism to leverage. In this case, a more direct approach is to force a checkpoint after the threads synchronize, eliminating the need to log interrupts.

Example of a System

[0057] **Figure 9** is a block diagram of an electronic system that can provide an environment for multithreaded processors. The electronic system illustrated in Figure 9 is intended to represent a range of electronic systems. Alternative electronic systems can include more, fewer and/or different components.

[0058] Electronic system 900 includes bus 910 or other communication device to communicate information, and processor(s) 920 coupled to bus 910 to process information. Electronic system 900 further includes random access memory (RAM) or other dynamic memory as well as static memory, for example, a hard disk or other storage device 935 (referred to as memory), coupled to bus 910 via memory controller 930 to store information and instructions to be executed by processor(s) 920. Memory 935 also can be used to store temporary variables or other intermediate information during execution of instructions by processor(s) 920. Memory controller 930 can include one or more components to control one or more types of memory and/or associated memory devices. Electronic system 900 also includes read only memory (ROM) and/or other static storage device 940 coupled to bus 910 to store static information and instructions for processor(s) 920.

[0059] Electronic system 900 can also be coupled via bus 910 to input/output (I/O) interface 950. I/O interface 950 provides an interface to I/O devices 955, which can include, for example, a cathode ray tube (CRT) or liquid crystal display (LCD), to display information to a computer user, an alphanumeric input device including alphanumeric and other keys and/or a cursor control device, such as a mouse, a trackball, or cursor direction

keys. Electronic system 900 further includes network interface 960 to provide access to a network, such as a local area network, whether wired or wireless.

[0060]   Instructions are provided to memory 935 from a storage device, such as magnetic disk, a read-only memory (ROM) integrated circuit, CD-ROM, DVD, via a remote connection (e.g., over a network via network interface 960) that is either wired or wireless, etc. In alternative embodiments, hard-wired circuitry can be used in place of or in combination with software instructions. Thus, execution of sequences of instructions is not limited to any specific combination of hardware circuitry and software instructions.

Conclusion

[0061]   Reference in the specification to "one embodiment" or "an embodiment" means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the invention. The appearances of the phrase "in one embodiment" in various places in the specification are not necessarily all referring to the same embodiment.

[0062]   In the foregoing specification, the invention has been described with reference to specific embodiments thereof. It will, however, be evident that various modifications and changes can be made thereto without departing from the broader spirit and scope of the invention. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.